CUDA implementation of Parallel Ford-Fulkerson Algorithm for Maximum Flow Problem

CS16B011, CS16B029, CS16B107

April 22, 2019

1 Introduction

A flow network or transportation network is a weighted directed graph in which edges represents a pathway for the flow and the weight represents the capacity of the link. The flow originates from a source and drains at the sink. At all other nodes the incoming flow must equal the outgoing flow. A flow network can be used to model traffic in computer networks, currents in electric circuits, fluids transportation etc.

Lester R. Ford and Delbert R. Fulkerson proposed the Ford-Fulkerson algorithm, a greedy algorithm to compute the maximum flow in a flow/transportation network. The core of the algorithm is to keep updating the flow in the network as long as it finds augmenting paths. We are going to implement a parallel version of the Ford-Fulkerson Algorithm [1] in CUDA.

2 Maximum Flow Problem Definition

Given a directed graph $\mathbf{G = (V,E)}$ with source **s** and sink **t**. The directed graph **G** is assumed to have no self loops, only one directed edge between a pair of nodes, no incoming edges to the source and no outgoing edges from the sink. The **capacity** of the edge $\mathbf{c(u,v)}$ represents the maximum flow through the edge **(u,v)**. The flow from u to v denoted by $\mathbf{f(u,v)}$ should satisfy the following two constraints:

1. $\mathbf{f(u,v)} \leq \mathbf{c(u,v)}$, for each $\mathbf{(u,v)} \in \mathbf{E}$. Capacity constraint.
2. $\forall\ \mathbf{v} \neq \mathbf{s,t}\ \sum_{(u,v)\in E} \mathbf{f(u,v)} = \sum_{(v,w)\in E} \mathbf{f(v,w)}$. Flow Conservation.

The net flow F through the network is defined as:

$$F = \sum_{(s,v)\in E} \mathbf{f(s,v)} = \sum_{(v,t)\in E} \mathbf{f(v,t)} \tag{1}$$

(i.e) The net outgoing flow from the source or the net incoming flow to the sink.

Given the input flow network, the goal of the problem is to find the maximum flow f satisfying these constraints with minimum runtime.

3 Available Resources

[1] discusses a version of the Ford-Fulkerson algorithm which can be parallelized. It also gives some details as to how we could go about parallelizing the algorithm. They have also mentioned about their simulation results and how much speedup they have got for various kinds of testbeds. We are planning to refer the paper for ideas on parallelization and to compare our speedup with what they have achieved.

Ford-Fulkerson algorithm being a fairly well known algorithm has a lot open source serial implementations which we are planning to use to verify the correctness of our implementation.

There is also a parallel implementation of the algorithm on GitHub. But since it doesn't use the same version of Ford-Fulkerson that we are planning to implement there isn't much to refer to in their implementation. The GitHub repository maintains a residual graph data structure whereas we are following a labelling based implementation.

4 Initial Ideas on Parallelization

Consider the part of algorithm where an augmenting path is to be found. Let a node be called labelled if the max flow which can pass through it (potential) is known and let it be called scanned if all it's neighboring nodes are also labelled.

In the serial version while scanning a node all it's neighbors which are unlabelled are updated sequentially. The idea is do the updates parallelly for not just a particular node but all nodes which are labelled and unscanned.

But in the above method, the pairs of nodes which are labelled and unlabelled respectively have to computed sequentially. This process can be avoided by going through all edges parallelly and checking for the above condition before updating the potential. This way even the limiting sequential step can be eliminated. Hence the implementation can potentially be sped up by the total number of edges.

Since there are few other parts to the algorithm, they can be tried to be done parallelly using CUDA streams. Concepts such as tiling might be difficult to introduce due to the non geometric nature of the input graph. Also we have to find efficient ways of using the shared memory per block for larger graphs.

5 Test Case Generation

The test bed was generated for different values of number of nodes ranging from 10 to 1000 and the number of edges randomly chosen from 0 to max number of edges possible. It was made sure that there is no incoming edges to the source, outgoing edges from the sink only one edge between a pair of nodes. The capacities of the links in a particular network were randomly chosen from 1 to a max value which can be set.

6 Serial Code Implementation and Profiling

The serial version of Ford-Fulkerson algorithm using the labelling method was implemented as mentioned in the paper [1]. Correctness of the implementation was verified by comparing the results on the testbed with an open source serial implementation [2], modified as per our input format. We profiled our code on a CPU with 1000-node networks and the results obtained are shown in Table 1.

Number of Nodes	*Number of Edges*	*Execution Time (seconds)*
1000	23777	0.211412
1000	78788	0.295294
1000	223639	0.381040
1000	362871	1.061324
1000	433746	1.500974

Table 1: Complete execution time for different graphs

Number of Nodes	*Number of Edges*	*Execution Time (seconds)*	*% of Total Execution*
1000	23777	0.191512	90.58%
1000	78788	0.257831	87.31%
1000	223639	0.325355	85.38%
1000	362871	0.893728	84.20%
1000	433746	1.281075	85.34%

Table 2: Sum of execution times for different graphs on parallelizable traversal components

In the serial code for a labelled, unscanned node each of it's neighbours are checked if they can be labelled in a sequential for loop. From the tables it can be seen that this part of the code(that can be parallelized) is taking up significant fraction of the complete execution time. Thus instead of iterating through all neighbours for each node sequentially, all edges are parallelly verified if they can be used to update. Using the mentioned idea we have made a first attempt at the parallel implementation.

7 Parallel Code Implementation and Profiling

7.1 Version 1

The implementation of the parallelizable part mentioned above was performed, where updation of labels was tried to be done for all edges parallelly. The number of blocks and number of threads per block were both initialized to the number of nodes in the graph. This allowed the updation of label of node u to scanned easier as all the threads(neighbours) in the block can be synchronized at the end. This also gave better performance when compared to other thread and block configurations since burst accesses can be availed for the variables in device's global memory.

The results on a testbed with 1000 nodes have been shown in Table 3. It can be seen that the time taken by the parallelizable part is much less in the gpu when compared to it's implementation on the cpu. But since the label array is frequently copied from host to device and vice versa it takes

up significant amount of the time. Moreover even the flow and capacity arrays have to copied from host to device.

Due to the above reasons, for testcases with smaller number of edges, the total execution time shoots up. But for testcases with larger number of edges, the speedup obtained on the gpu predominates and gives a speedup compared to the cpu.

Number of Nodes	*Number of Edges*	*GPU Execution Time (s)*	*TotalTime (s)*	*SpeedUp*
1000	23777	0.00898208	0.275961	0.767
1000	78788	0.022601114	0.391236	0.754
1000	223639	0.046606575	0.619192	0.615
1000	362871	0.088054253	0.949427	1.12
1000	433746	0.104566742	1.144858	1.31

Table 3: Execution times for parallelized Ford Fulkerson

7.2 Version 2

The context switch from cpu to gpu and vice versa and copying of the label array for the same can be avoided if the entire augmenting path from source to sink is found on the kernel. And the parallelizable part from above can be called from the kernel.

The results on the same testbed with 1000 nodes have been shown in Table 4. It can be seen that the time executed on the gpu is greater than that in version 1. This is since all the computation including updating the flows is done on the gpu. This implementation avoids the frequent copy of memory between device and host. But still introduces a new overhead in terms of calling a new kernel from the gpu.

For testcases with smaller number of edges, the total execution still remains higher than that on the cpu possibly due to the overhead introduced by copying flow and capacity arrays in smaller graphs as they predominate the computation. But for testcases with larger number of edges, the speedup obtained on the gpu predominates and gives a speedup compared to the cpu. Moreover the speedup obtained in version 2 is greater for all testcases than that in version 1.

Number of Nodes	*Number of Edges*	*GPU Execution Time (s)*	*TotalTime (s)*	*SpeedUp*
1000	23777	0.041868256	0.268674	0.786
1000	78788	0.12127874	0.330214	0.894
1000	223639	0.253095123	0.491654	0.775
1000	362871	0.502569611	0.757221	1.40
1000	433746	0.579041199	0.821741	1.83

Table 4: Execution times for parallelized Ford Fulkerson with Dynamic Parallelism

References

[1] Zhipeng Jiang, Xiaodong Hu, and Suixiang Gao. A parallel ford-fulkerson algorithm for maximum flow problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 70. The Steering Committee of The World Congress in Computer Science, Computer ..., 2013.

[2] Ford-Fulkerson Algorithm for Maximum Flow Problem. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem.